Programmation orientée objet

Benoit Damiani, Matthieu De Pillot De Coligny

Rapport de projet myUber

06/12/2018

Résumé
L'objectif de ce projet était dans une première partie de reproduire le fonctionnement de Uber sous Java, en prenant en compte les différentes exigences de flexibilité du programme. Puis dans une seconde partie, fournir une interface graphique permettant de gérer l'environnement Uber en temps réel.

CentraleSupélec

Table des matières

1 myUber core

1.1 GPS

Au vu des nombreuses positions GPS à gérer, nous avons créé une classe GPS pour gérer un couple de flottants et les opérations associées. On peut ainsi trouver dans cette classe deux méthodes statiques : une pour calculer la distance entre deux points, l'autre pour répondre au problème du plus court trajet pour les uberPool. Concernant cette dernière fonction, l'algorithme consiste à créer une liste des points que peut visiter le chauffeur à l'instant considéré, constitué au début des positions des clients à récupérer. Cette liste évolue en ajoutant les destinations des clients que l'on a récupéré et en retirant les points visités par le chauffeur, en choisissant toujours la plus courte distance.

1.2 Rides

Pour les rides, il est demandé que de nouveaux types de rides puissent être ajoutés facilement. Pour répondre à cette problématique, nous avons adopté un design de simple factory, en implémentant donc une classe RideFactory et une classe abstraite Ride dont laquelle hérite tous les types de Ride. La classe Ride contient toutes les fonctions nécessaires au calcul des prix et des durées de trajet, étant donné que toutes les variables nécessaires à ces calculs se trouvent dans cette classe.

Dans chaque classe correspondant à un type de ride, on initie dans le constructeur les deux listes correspondant aux tarifs appliqués sur ce type de ride et le type de voiture demandée. Cela permet de n'avoir à coder qu'une seule fois la fonction calculant le coût dans la classe Ride.

1.3 Car

Tout comme les rides, il est demandé de pouvoir ajouter des classes de voiture facilement à notre code, d'où la création d'une classe CarFactory afin de créer les voitures selon le type spécifié. La CarFactory a également d'autres avantages : nous nous en servons pour recenser toutes les voitures disponibles actuellement dans myUber ainsi que le nombre de chaque type. Ainsi pour créer un nouveau type de voiture il suffit de créer une nouvelle classe qui hérite de Car ainsi que son constructeur, et d'ajouter la condition de création dans la CarFactory. Comme une voiture peut être conduite par plusieurs driver, nous avons pris la liberté d'ajouter deux attributs : un état "available" ou "non-available", et une liste des driver associés.

1.4 Driver et Customer

Nous avons décidé de créer une classe DriverFactory selon un Factory pattern afin de simplifier la lecture des autres codes lors de la création d'un driver, ainsi que de sauvegarder sous forme de liste modifiable tous les objets driver, et leur nombre. L'objet driver possède toutes les caractéristiques demandées, ainsi qu'une fonction d'évaluation qui n'est pour l'instant pas utilisée car il n'y a pas de réelle interaction avec un vrai client.

Nous avons mis en place les mêmes outils pour les customers que pour les drivers : un CustomerFactory pour simplifier, qui sauvegarde une liste de tous les objets Customer. La boîte mail des clients sera implémentée dans la partie 2.

1.5 MyUber

La classe MyUber est la classe principale, celle où s'exécute le main. La fonction d'exécution a été découpée en sous-fonction pour simplifier la lecture et la compréhension du code. On retrouve ainsi :

- Une fonction setup, qui permet de créer un nombre donné de voitures standards, de berlines et de vans, ainsi que leur conducteurs, et un nombre donné de clients.
- Une fonction RideAlea, qui permet de renvoyer un type de ride aléatoire, pour simuler le choix du client.
- Une fonction trouverVoiture, qui renvoie la voiture du type demandé disponible, dont au moins un des conducteurs est on-duty la plus proche. Si toutes les voitures du type demandé sont indisponibles, la fonction renvoie une référence vide.
- Une fonction trouverConducteur, qui renvoie un conducteur "on-duty" pour une voiture entrée en paramètre. Encore une fois s'il n'y en a pas, la fonction renvoie une référence vide.
- Une fonction faireUnRide, qui correspond au moment où le client entre physiquement dans la voiture. Elle sera expliquée plus tard.
- La dernière fonction ne fait partie du corps principal, il s'agit de la fonction getTraffic, qui permet de définir en fonction de l'heure réelle l'état du trafic de la simulation

La fonction main en elle-même n'apporte que la notion de file d'attente. Tous les clients sont dans une liste, et grâce à une boucle while la fonction ne s'arrête que lorsque tous les clients ont été satisfaits. Cela donc peut poser un problème si nous ne créons pas de berline dans le setup, par exemple, car des clients pourraient l'attendre indéfiniment. Le bookOfRides contient toutes les informations sur les trajets effectués. Celles-ci étant de types différent, elles sont stockées dans des ArrayList séparées dans la classe BookOfRides. Cela permet une manipulation plus aisée des données nécessaire au calcul des statistiques. Le calcul des statistiques est lui aussi contenu dans une classe dédiée mais cette fois pour des raisons de lisibilité.

1.6 Threading

Nous avons eu besoin d'utiliser des threads pour pouvoir gérer le fait que plusieurs clients commandent en même temps, et qu'il soit possible d'annuler une commande. Nous avons choisi l'option du timerTask, que nous avons jugé comme convenant à notre programme. Nous avons deux classes d'objet TimerTask : DriverToCustomer, qui représente le temps de trajet pour que le driver rejoint le customer ; et la classe RideEnCours qui représente le temps du trajet entre le point de départ du client et son point d'arrivée. D'un point de vue programme, ces deux classes diffèrent peu : lorsqu'on est arrivé au bout du timer, les deux mettent à jour les GPS et les statuts des entités en jeu. La seule différence est la prise en compte du cas où le client peut annuler avant que le driver n'arrive (donc annuler le thread).

Le changement des statuts des objets dès le lancement des actions permet d'être sûr de ne jamais être dans une situations de deadlocks.

1.7 Statut des objets

Nous avons pris quelques libertés en ajoutant des statuts aux objets que nous manipulons, afin de rendre plus efficace notre programme, surtout vis-à-vis de la GUI. Ils sont exposés ici pour les différents objets concernés :

Pour le driver, il est :

— "on-duty" s'il est disponible
— "off-duty" lorsqu'il prend une pause, et "offline" lorsqu'il est déconnecté
— "asked" lorsqu'une ride lui est proposée (remarque : il ne peut pas refuser)
— "drivingToC" lorsqu'il est en route pour rejoindre un client
— "cancelled" si le client annule sa commande (cela permet de notifier le driver)
— "waiting" lorsqu'une fois sur le lieu de rendez-vous il attend le client
— "on-a-ride" lorsqu'il conduit le client
— "end_ride" lorsque la ride est terminée (encore une fois pour le notifier)

La voiture est :

— "available" si elle n'est pas engagée dans un ride
— "unavailable" si elle est engagée dans un ride

Le client est :

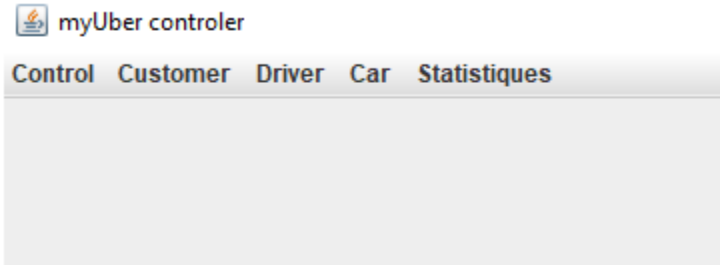
— "free" s'il n'est pas engagé dans un ride (cela comprend également la phase de commande)
— "waiting_confirmation" lorsque la commande est passée, et en attente d'acceptation par le driver
— "waiting_driver" lorsque la commande est prise, et que le driver se déplace vers le point de départ

— “driver_arrived” lorsque le driver est arrivé au point de départ, et qu’il attend le client
— “on-a-ride” durant le trajet
— “asked_eval” une fois le ride terminé (afin de lui proposer une évaluation)

2 Interface graphique (GUI)

2.1 Utilisation du programme

Pour accéder à l’interface il faut lancer la fonction main de la classe MyUber. Une première fenêtre s’ouvre pour demander le scénario (nombre de voitures et type, nombre de clients, le système va automatiquement créer et associer un driver par voiture). On accède ainsi à la fenêtre principale, représentée ci dessous :



Il y a cinq menus déroulants, que je vais expliquer en détail ici.

2.2 Le menu Control

Le menu control possède deux items, le premier permet de faire des ajouts (item Ajouts), conformément au cahier des charges c’est à dire :
— créer un client à partir d’un nom et prénom
— créer un driver et une voiture à partir d’un nom, d’un prénom et d’un type de voiture
— créer un driver seul à partir d’un nom et prénom, et lui associer une voiture déjà existante

Le second item est l’item déplacement, qui permet de déplacer manuellement et instantanément une voiture ou un client, juste en entrant de nouvelles coordonnées GPS.

2.3 Le menu Car

Ce menu est le plus simple : il possède autant d’Item que de voiture en ce moment dans le système, et affiche pour chaque voiture ses caractéristiques :

— type
— état
— coordonnées
— liste de ses conducteurs

2.4 Statistiques

Le menu statistiques possède trois items, qui correspondent aux statistiques des drivers, des customers, et aux statistiques globales. Chaque onglet permet d'accéder aux fonctions demandées dans l'énoncé du projet.

2.5 Le menu Driver

Le menu driver possède autant d'item que de drivers dans le système à l'instant considéré. Chaque item correspond à la page d'un driver, telle qu'il pourrait l'avoir sur son application. Celle-ci est séparée en trois panels : le premier représente ses caractéristiques (nom, prénom, état, évaluation). Le second est son "garage", c'est-à-dire les voitures qu'il peut conduire. Le troisième est le panel d'action, c'est par celui là qu'il prend toutes ces décisions. Lorsqu'il n'est pas concerné par un ride, c'est avec ce panel qu'il peut alterner entre les états "on-duty", "off-duty" et "offline".

2.6 Le menu Customer

Forgé sur le même style que le menu Driver, celui-ci possède encore autant d'item que de customers à l'état actuel, et chaque customer correspond à un item. La page d'un customer est elle composée de deux panels, le premier est un récapitulatif de ses informations (nom, prénom). Le second est le panel d'action, que lui permettra de prendre les décisions lors de la commande d'un ride, que nous verrons dans le point suivant.

2.7 Réaliser un ride

La réalisation d'un ride via la plateforme GUI se structure en différentes étapes :

Etape 1 : le customer fait la commande en commençant par "rechercher un ride" sur son écran d'action, puis en entrant les coordonnées GPS de sa destination. Le système lui renvoie les prix des différents ride, et il peut en choisir un. Si ce ride n'est pas disponible (plus de voiture du type ou de driver disponible), le système envoie un message d'excuse, et le customer est renvoyé à sa page "d'accueil".

Etape 2 : si la ride est disponible le système contacte le driver concerné pour lui demander de l'accepter (il est obligé, mais doit faire l'action). Le customer a un message

d'attente, ainsi qu'un bouton "annuler" pour tout annuler à tout moment. Une fois que le driver accepte (le client est notifié de l'acceptation), il se dirige vers le point de départ où il retrouve le client.

Etape 3 : lorsque le client a rejoint le driver, il lance le ride. A partir de là plus d'annulation possible. Les deux sont en état "en route".

Etape 4 : à l'arrivée le driver est notifié, le système propose au customer une évaluation, qu'il peut éviter. Le système revient à l'état de repos.

Il est bien sûr possible d'effectuer plusieurs rides en même temps ! Dans un souci d'utilisation à notre échelle du programme, nous avons fixé l'ordre de grandeur d'un trajet autour de la minute (il dépend bien évidement de la distance et du trafic à l'heure actuelle, réelle).

3 Organisation du travail

Afin de construire l'ensemble du programme, nous avons réparti les différentes parties à chaque membre de notre binôme. Les classes Car et Ride par exemple peuvent être codées relativement séparément. D'autre part, afin de disposer toujours de la version à jour du programme, nous avons eu recours à GitHub qui permettait en plus d'obtenir un résumé des actions de l'autre par les commits.

Du point de vue du développement, notre approche a été celle d'un "test-driven development" avec la création de plusieurs fonctions JUnit pour tester notre programme au fur et à mesure des modifications.

La répartition des tâches dans notre groupe peut se représenter de la façon suivante :

Tâche réalisée	UML	Ride	Car	Driver	Customer	Multi-threading	GUI	Statistiques
Benoit	X	X			X			X
Matthieu			X	X		X	X	

Notre travail en binôme a été rendu plus compliqué lors de la deuxième partie car l'ordinateur de Benoit a dû être envoyé en réparation, et toutes les modifications du code ne pouvaient donc se faire plus que sur un seul ordinateur.

4 Pistes d'amélioration et conclusion

A l'issue de cette deuxième partie, nous n'avons pas pu implémenter UberPool dans notre programme. Un autre défaut est que la gestion des threads a été couplée avec l'interface graphique : dans un souci de flexibilité il faudrait reprendre notre solution pour séparer plus nettement les deux. Enfin, la gestion des erreurs n'a pas été réalisée, ce qui peut par exemple poser des problèmes si l'on entre des valeurs aberrantes dans l'interface graphique. La version finale de notre projet ne répond pas à toutes les demandes de l'énoncé (notamment le cas d'UberPool), mais respecte le principe "open-close" concernant l'ajout de nouveaux types de Car ou de Ride tout en permettant une utilisation simple grâce à l'interface graphique et l'utilisation de threads pour gérer l'état de l'ensemble des rides.